SeCoGen – A Service Code Generator

Ricardo Queirós
ESMAD – Polytechnic of Porto, Portugal
CRACS – INESC TEC, Porto, Portugal
http://www.ricardoqueiros.com
ricardoqueiros@esmad.ipp.pt

Abstract

The architectural pattern of micro-services is being increasingly adopted by developers, facilitating the maintenance and scalability of the systems' code. The adoption and consumption of these micro-services are often seen on the front-end code of the Web applications. Nevertheless, this adoption obliges web designers/developers to know where to look for those web services, to read their documentation and to write the request/response code as well to control the corresponding UI rendering. This whole process is time-consuming and error-prone. This article introduces SeCoGen as an interactive code generator for Web service parsing and consumption. The generator benefits from an HTTP request template, a query normalizer and dynamic UI templates. In order, to validate the generator feasibility and usefulness, a REST API to search for countries is used.

2012 ACM Subject Classification Software and its engineering → Source code generation

Keywords and phrases Code Generation, Web services, micro-services

Digital Object Identifier 10.4230/OASIcs.SLATE.2019.23

Funding This work is financed by National Funds through the Portuguese funding agency, FCT – Fundação para a Ciência e a Tecnologia within project: UID/EEA/50014/2019.

1 Introduction

The Web is evolving at a fascinating rate. One of the reasons for its success is a large number of sophisticated frameworks to create responsive, adaptive systems with very high levels of performance. At the same time, the architectural pattern of micro-services has increasingly been adopted in Web development [1], enabling developers to write components using different languages, to facilitate system scalability, and to simplify the deploy of separate system components, among others. However, this architecture also brings a number of challenges. For example, spreading log information and tracking transactions that represent a single context, but are performed across multiple services. Another major difficulty is the search for the most appropriate services, the reading of its documentation (when it exists) and the coding of the request-response circuit and its respective visual rendering in the user's browser. To make things worse, the developer still has to worry about using code that is compatible with the several existent browsers supported by the market.

This article presents SeCoGen as a Web service code generator. The generator acts as an interactive environment organized in 3 steps: the developer 1) searches and selects the desired API; 2) selects the desired endpoint and, if necessary, enters the respective input parameters; 3) choose the template for the response rendering. After these steps, SeCoGen builds and bundles all the code needed to handle the Web service request/response circuit.

In order to evaluate SeCoGen usefulness and feasibility, the environment is tested using the countries API. The main goal is to produce a responsive Web catalog for European countries listing.

The remainder of this paper is organized as follows: Section 2 reviews the current code generation tools and techniques. In Section 3, we present SeCoGen and describe its two use cases: service register and service code generation. In Section 4, we validate the SeCoGen

environment enumerating the steps for the generation of a responsive Web catalog based on the Countries API. Finally, we conclude with a summary of our main contributions and a perspective of future research.

2 Code Generation

Code generation is typically defined as the process by which a compiler's code generator transforms an intermediate representation of source code into other forms of code which can be readily executed by a machine [5].

In the development realm, code generation can be a very important approach to boost productivity in the development process. Regardless of the code generation context, it fosters productivity, simplification, and consistency. In practice, using this approach, the developer writes the generator once and it can be reused as many times as he needs making it significantly faster than writing the code manually. Another important aspect is simplification since you typically generate your code from some abstract description. It means that your source of truth becomes that abstract description and not the code. That description is typically easier to analyze and check compared with the whole generated code [4]. Finally, with code generation, you get always the code you expect. The generated code is designed according to the same principles. With the code written manually instead you can have different developers using different styles and often introducing errors even in the most repetitive code.

There are several types of generation tools available such as: template engines, parser generators, ad-hoc applications and model/database driven design tools. In the following subsections, we detail the first three approaches.

2.1 Template engines

A template engine is basically a compiler that can interpret a template file containing special notations written in a simple template language. The simplest thing it can do is replacing this special notation with the proper data, given at runtime.

The majority of the template engines also supports other programming constructs such as simple flow control commands (e.g., for-loops, if-else-statements). There are several template engines out there. The most notable examples are Pug, Mustache, HandleBars, and EJS.

Pug (formerly known as "Jade") is a high-performance template engine influenced by Haml and implemented with JavaScript for Node.js and browsers. Pug is available via npm. Its rendering process is done by compiling the Pug source code into a JavaScript function that takes a data object as an argument. Finally, calling that function will return an HTML string rendered with the respective data.

Mustache and *Handlebars.js* are both logic-less templating languages which keep the view and the code separated. Both work by expanding tags in a template using values provided in a hash or object.

EJS is a very simple templating language that allows the generation of HTML markup with plain JavaScript. One of its distinguished features is the chance to download a browser build from the latest release and use it in a script tag.

2.2 Parser Generators

A parser generator is a tool that helps you to create parsers. A parser receives a piece of text and transforms it into an organized structure, such as an Abstract Syntax Tree (AST).

The parser generator will produce a new language programming source files (e.g. Java, C/C++, C#, Python, Ruby) which can be divided into several cooperating modules:

- *Lexer*: reads an input character or byte stream (i.e. characters, binary data), splits it into tokens using specified patterns, and generates a token stream as output.
- *Parser*: reads a token stream (often generated by a lexer), and matches phrases in your language via the specified patterns, and typically performs some semantic action for each phrase (or sub-phrase) matched. Each match could invoke a custom action, or generate an Abstract Syntax Tree (AST) for additional processing.

A parser generator takes a grammar (formal description typically written in EBNF format) as input and automatically generates source code that can parse streams of characters using the grammar. The generated code is a parser, which takes a sequence of characters and tries to match the sequence against the grammar [2].

Nowadays, there are several parser generators. The most notable examples are ANTLR, JavaCC and Flex.

ANTLR, ANother Tool for Language Recognition, is a tool which receives a grammar and generates source code files and other auxiliary files [4]. In practice, using ANTLR there are the following set of tasks:

1. define a lexer and parser grammar
2. invoke ANTLR: it will generate a lexer and a parser in your target language (e.g., Java, Python, C#, Javascript)
3. use the generated lexer and parser: you invoke them passing the code to recognize and they return to you an AST

2.3 Ad-Hoc Applications

Other tools were not mentioned since they are not suitable in the presented categories, but their huge importance obliges to create this category such as the scaffolding tools Yeoman, Slush and Lineman. These tools have something in common: they simplify the life of developers, by providing a way to create ready-to-use project optimized for a particular software platform, library or need.

Yeoman is a very popular and generic scaffolding system which generates with one command a new project that automatically implements all the Web best practices. The core of Yeoman is a generator ecosystem, on top of which developers build their own templates (thousands of templates available). Yeoman is written in JavaScript, so developing a generator requires simply to write JavaScript code and using the provided API. The workflow is also very simple: you ask the user information about the project (e.g., its name), gather configuration information and then generate the project [3].

Regarding GitHub stats, Yeoman takes the lead with 9.1K stars and 765 forks against 1.2K of Lineman with 89 forks. The Slush project has low values.

3 SeCoGen

SeCoGen is a Web environment for generating Web service code, namely: the code for the HTTP request (method, URL target, parameters, and body) and all the code needed to handle the response, from the parsing of the service response to the User Interface (UI) rendering.

The generator has two use cases: the service registration and the service code generation. In the former, developers will select the API service and its endpoints, define the placeholders for the input request parameters and define the queries to normalize the response data

according to supported UI templates. In the later, it is expected that anyone, including people without any programming knowledge, to use SeCoGen to automatically generate the service code in simple steps. The following subsections detail both use cases:

3.1 Service Registration

In order to be able to generate the code for a Web service, it must be properly configured and stored in a database. The architecture of SeCoGen service registration is depicted in Figure 1:

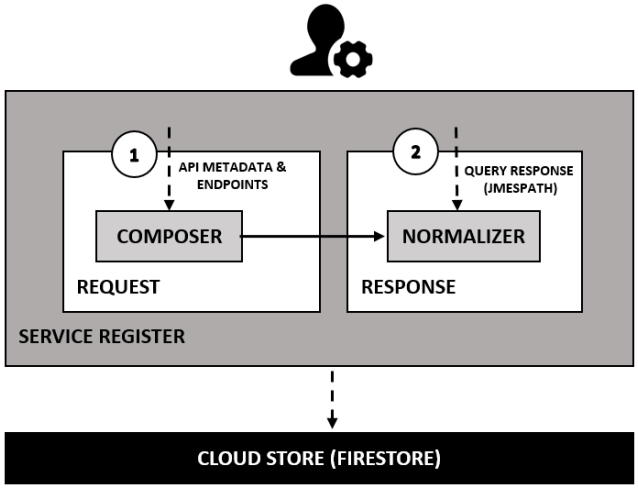

Figure 1 Service registration.

The service creation requires the admin/developer to fill several fields (1): the name of the API, its metadata (API description, main link, tags, etc.) and a list of HTTP messages supported by the API (commonly called of endpoints).

An HTTP request message includes a simple structure and is composed by:

- an HTTP method, a verb (like GET, PUT or POST) which describes the action to be performed;
- the request target, usually a URL;
- an (optional) body. Some requests send data to the server in order to update it: as often the case with POST requests (containing HTML form data).

The following code excerpt shows a generic HTTP message request template:

Listing 1 HTTP request message register.

```
-- HTTP request method and target --
GET http://api.endpoint.com/$1?q1=$2
-- HTTP body (optional) --
field=$3
...
field=$N
```

Both the request URL and body have placeholders that must be described during the registration process. Each placeholder will then be filled by the consumer of the service in order to generate an actual request.

After formalizing the request, it is necessary to analyze the type of response of the service and what is going to be presented visually to the user (2). Thus, SeCoGen provides a configurable template-based rendering system. At the moment, three templates are configured:

- Selector - displays the response of a Web service in an HTML selector. It is the ideal solution for secondary services where the result will be used in a larger process (e.g., web form)
- table - displays the response in tabular format with N columns. It is the ideal scenario to display a list of items, where each item has several textual characteristics (e.g., list of items to buy in an online store)
- Card - displays the response on a responsive card composed by data from different types, such as images, text, and buttons. Ideal for displaying catalogs (e.g., a store's product catalog)

A template is encoded in HTML/CSS. This separation of the request/response code from the UI render code is fundamental to the modularization and scalability of the system. At the same time, it enhances the contribution of different types of users in the generator. In this case, the templates will be fed into the system by Web designers. Here's an excerpt from a Bootstrap Card template to format any service response:

Listing 2 UI response template.

```
<div class="card">
  <img src="$image" class="card-img-top">
  <div class="card-body">
    <h5 class="card-title">$title</h5>
    <p class="card-text">$subtitle.</p>
    <a href="$linkValue" class="btn btn-primary">more details</a>
  </div>
</div>
```

In order to map the actual response of service to one of the supported templates, it is necessary, in the service registration phase, to define a query that will filter the response data into a normalized format supported by the SeCoGen templates. For this normalization process, we use JMESPath as a query language for JSON. The JMESPath language is described in an ABNF grammar with a complete specification. This ensures that the language syntax is precisely defined. The following excerpt shows the mapping between a random service response and the normalized format supported by SeCoGen templates:

Listing 3 Service response normalization.

```
-- Example of a service response --
product:
[
 {
  img: "p1.jpg",
  product: "HP 15-DA0036NP",
  metadata: {desc:"A great HTP laptop", target:"hp15-DA0036NP.html"},
  stars: "3"
 },
 ...
]
-- JMESPath query --
"product[*].{
    scg_image: @img,
    scg_title: @product,
    scg_subtitle: @metadata.desc,
    scg_link: @metadata.target
}"
```

```
-- Normalized JSON --
[
 {
  "scg_image": "p1.jpg",
  "scg_title": "HP 15-DA0036NP",
  "scg_subtitle": "A great HTP laptop",
  "scg_link": "hp15-DA0036NP.html"
 },
 ...
]
```

3.2 Service Code Generator

With services properly configured and stored in the database, users can access SeCoGen to generate the desired Web service code. The architecture of this use case is presented in Figure 2:

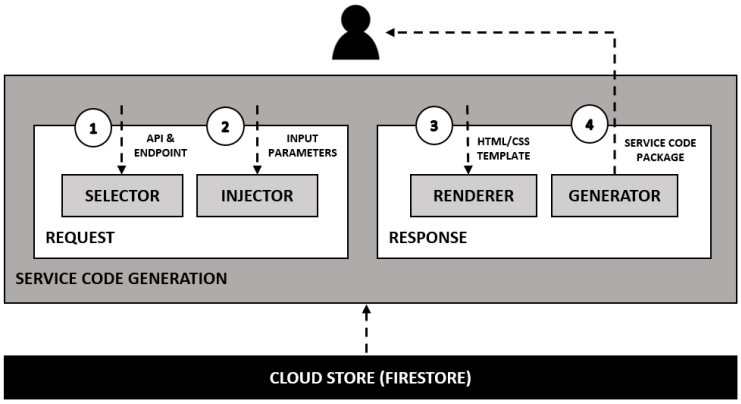

Figure 2 Service register.

The service code generation process is iterative and is organized in two phases: generation of the HTTP request and generation of the response parsing and its rendering. The user begins by selecting the desired API through a search based on tags (1). Next, all the endpoints of the selected API are listed. After choosing an endpoint and, if the endpoint has configurable parameters, the user is invited to define values for all of them (2). In a second step, the user chooses the template (3) that he intends to use to display the service response data (selector, table, or card).

The generator will normalize the service response to the SeCoGen required format and benefit from all the templates presented in the previous subsection to build a package with all the necessary files to represent the request/response circuit of the selected service. The following excerpt shows, in a high level of detail, how all this process is executed:

Listing 4 UI generation based on templates.

```
import * as jmespath from "./jmespath.js"
const responseData = ...
const normalizedJSON = jmespath.search(responseData, @JMESPathQuery)
Secogen.map(normalizedJSON, template)
```

3.3 Data Model

SeCoGen uses a real-time NoSQL database in the cloud called FireStore. Cloud Firestore is a flexible and scalable database for the mobile device, Web and server development from the Firebase and Google Cloud Platform. Like the Firebase Real-time Database, it keeps your data in sync in client applications through real-time listeners. In addition, it offers offline support for mobile and Web devices so you can build responsive applications that work regardless of network latency or Internet connectivity.

The data model of Cloud Firestore is quite flexible. The data is stored in documents that contain field mappings for values. These documents are stored in collections, which are document containers that are compatible with many different data types, from strings and simple numbers to complex and nested objects. You can also create sub-collections within documents and create hierarchical data structures that can be scaled as the database grows.

For the SeCoGen context two collections were defined:

- apis - stores information about APIs and its endpoints. For each endpoint it stores information about the HTTP request message, the conversion query, and other metadata;
- templates - stores all rendering templates which must be independent of the services and their implementation languages.

4 Validation

In order to evaluate SeCoGen usefulness and feasibility, the environment was tested using the countries API [1]. The Countries API allows to get information about countries through a very simple RESTful API.

In the service registration process, the API and some of the most important endpoints were added. In this case, the most important endpoint is the one that lists information from all countries by region. Then a query was defined to map the response with the Card template. On the generation side of the service, first, the user selected the API and its respective endpoint and injected EU (Europe) as the input parameter. Finally, the Card template was selected. The next excerpt shows the main code snippets:

Listing 5 Normalization code for the Countries REST API.

```
-- HTTP message request template
https://restcountries.eu/rest/v2/regionalbloc/$1
-- Normalization query --
"country[*].{
    scg_image: @flag,
    scg_title: @translations.pt,
    scg_subtitle: "Capital: @capital",
    scg_link: "https://pt.wikipedia.org/wiki/@translations.pt"
}"
```

After SeCoGen generated all the required code, the user downloaded the zip file and ran the code in a browser. The final result is shown in Figure 3:

[1] Link: `https://restcountries.eu`

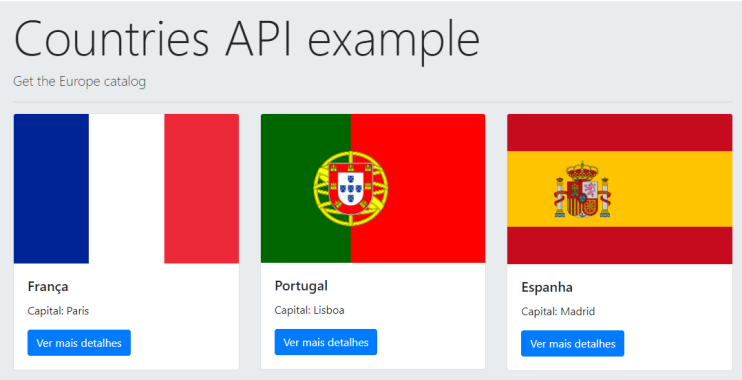

Figure 3 Countries API response render based on the Card template.

5 Conclusion

This article presents SeCoGen as a code generator for the request-response circuit of a Web service. Its ultimate goal is to help anyone to generate code for a Web service without any need of prior domain and programming knowledge. For this purpose, we have explained the service registration process, which entails filling all the details of an API and its endpoints and the code generation process, which consists of three simple steps: API choice, endpoint choice with the respective input parameters injection and the rendering template choice. As a result, SeCoGen generates all necessary files and packages all of them into a ZIP file that can be downloaded by the user and executed in any modern browser.

In order to validate the generator, an API was used to list country data as a responsive Web catalog. The validation process has proven that SeCoGen has enormous potential, yet there are several issues still to be addressed.

In the near future, the following tasks will be tackled:

- support for POST requests and creation of a template based on a web form
- creation of a back office in order to facilitate the registration process of services
- support for GraphQL (an alternative to REST architectures)

References

1 P. D. Francesco. Architecting Microservices. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 224–229, April 2017. doi:10.1109/ICSAW.2017.65.

2 Robert Miller and Max Goldman. Software Construction. https://ocw.mit.edu/ans7870/6/6.005/s16/classes/18-parser-generators/#reading_18_parser_generators, 2016. Spring 2016. Massachusetts Institute of Technology: MIT OpenCourseWare.

3 Ricardo Queirós. PROud – A Gamification Framework Based on Programming Exercises Usage Data. *Information*, 10(2), 2019. doi:10.3390/info10020054.

4 Gabriele Tomassetti. A Guide to Code Generation. https://tomassetti.me/code-generation/, 2018. Accessed: 2018-11-01.

5 Seung-Su Yang, Hyung-Joon Kim, Nam-Uk Lee, and Seok-Cheon Park. Design of Automatic Source Code Generation Based on User Pattern Definition. In James J. Park, Vincenzo Loia, Gangman Yi, and Yunsick Sung, editors, *Advances in Computer Science and Ubiquitous Computing*, pages 1434–1439, Singapore, 2018. Springer Singapore.